Synopsis V1.0
Proton and Heavy Ion Single Event Effects Testing
of the Motorola ColdFire (MCF5307) Processor

Jim Howard[2], Austin Lanham[1], Lamar Dougherty[1], Linh Nguyen[1],
Joe Roman[1], and Tim Irwin[3],
1. NASA / Goddard Space Flight Center, Greenbelt, MD
2. Jackson and Tull Chartered Engineers, Washington DC
3. QSS, Inc. Lanham, MD

Test Date: June 6, 2002 & August 12, 2002 Report Date: April 11, 2003.

I. Introduction

This study was undertaken to determine the proton Single Event Effects (SEE) and total dose susceptibility of the Motorola ColdFire (MCF5307) Processor. The device was biased and operating when exposed to a proton beam at the Indiana University Cyclotron Facility (IU) and heavy ion beams at the Texas A&M University Cyclotron Single Event Effects Test Facility. The processor software was written to test various sections of the processor throughout the exposures.

II. Devices Tested

One part was exposed to protons and three devices were exposed to heavy ions for this testing. These device were manufactured by Motorola and was characterized prior to exposure. The lot date code for these devices was 0120.

III. Test Facilities

Facility: Indiana University Cyclotron Facility
Proton Energy: 189.9 MeV incident on DUT structure
Flux: 8.8 x 10^7 to 2.7 x 10^8 protons / cm^2 / s.

Facility: Texas A&M University Cyclotron Single Event Effects Test Facility
Flux: 4.9 x 10^3 to 8.8 x 10^4 particles / cm^2 / s.

Table I

Ion	Energy (MeV)	LET ($MeVcm^2/mg$)
Ne	264	2.8
Ar	496	8.7
Kr	942	28.9
Xe	1321	53.6

IV. Test Hardware and Software

The test setup consisted of a laptop PC for output collection, a 75-foot serial cable, a 75-foot ribbon cable and the M5307C3 evaluation board manufactured by Matrix Design & Manufacturing, Inc.

The code was downloaded via CodeWarrior IDE, a P&E Microsystems 'Wiggler' and the Background Debug Monitor (BDM) of the evaluation board. The test code was also stored in Flash ROM, but due to unknown linking problems, only portions of the code would execute properly. After the code was acquired from either Flash or the BDM, the code was stored and run from DRAM. The beam of radiation was targeted directly at the chip so that the peripheral devices were less likely to be harmed.

The MCF5307 was tested while running on the M5307C3 evaluation board. This board utilizes most of the functions of the ColdFire processor. The test setup consisted of a PC laptop, an HP6237B Power Supply, a Keithley 2000 multimeter, the M5307C3 evaluation board, a 75 ft. ribbon cable, a 75 ft. serial cable, and a 75 ft. power cable. The power supply was set to output 8V with 3A current limiting and the Keithley was used to monitor the current draw of the board. The ribbon cable was used to download code from the laptop to the board and the serial cable was used to output data from the board to the laptop. A block diagram of the board and its inputs and outputs is shown in Figure 1.

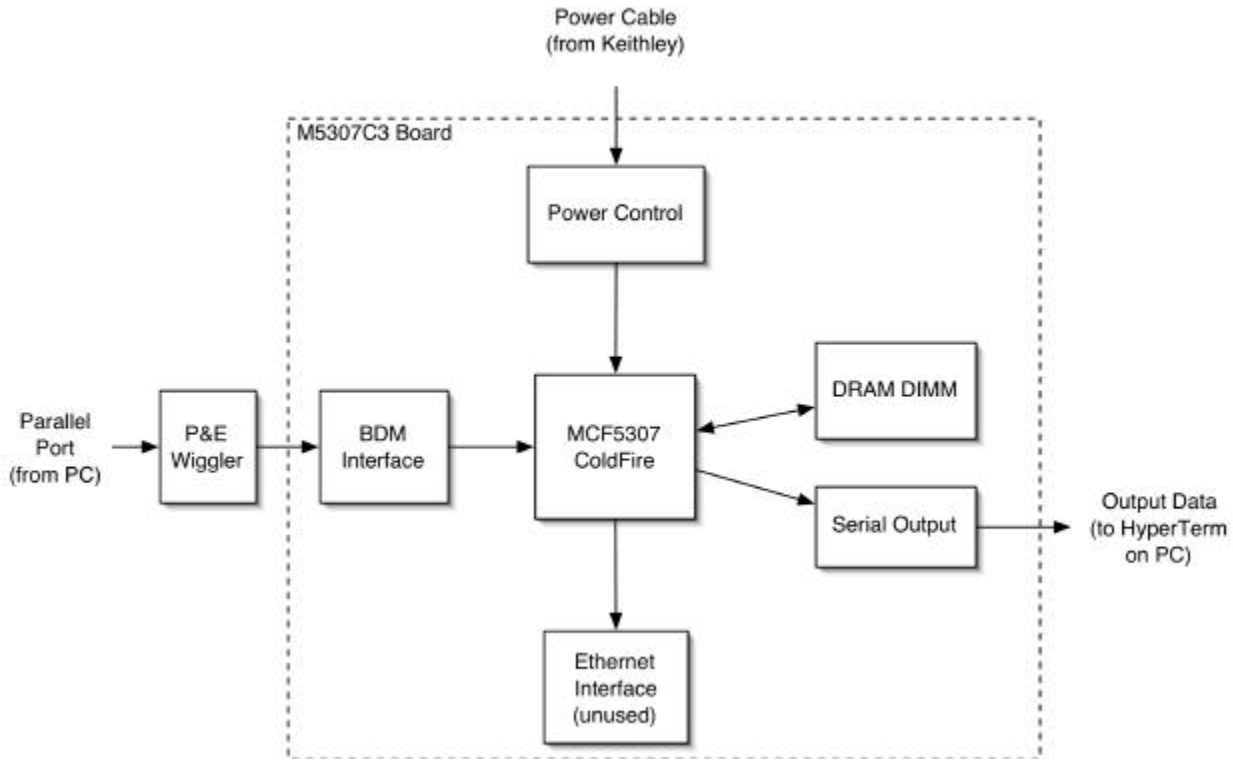

Figure 1. M5307C3 Eval. Board block diagram.

To fully test all of the functionality of the chip, a firm understanding of the chip's functionality is required. Figure 2 shows the MCF5307's block diagram.

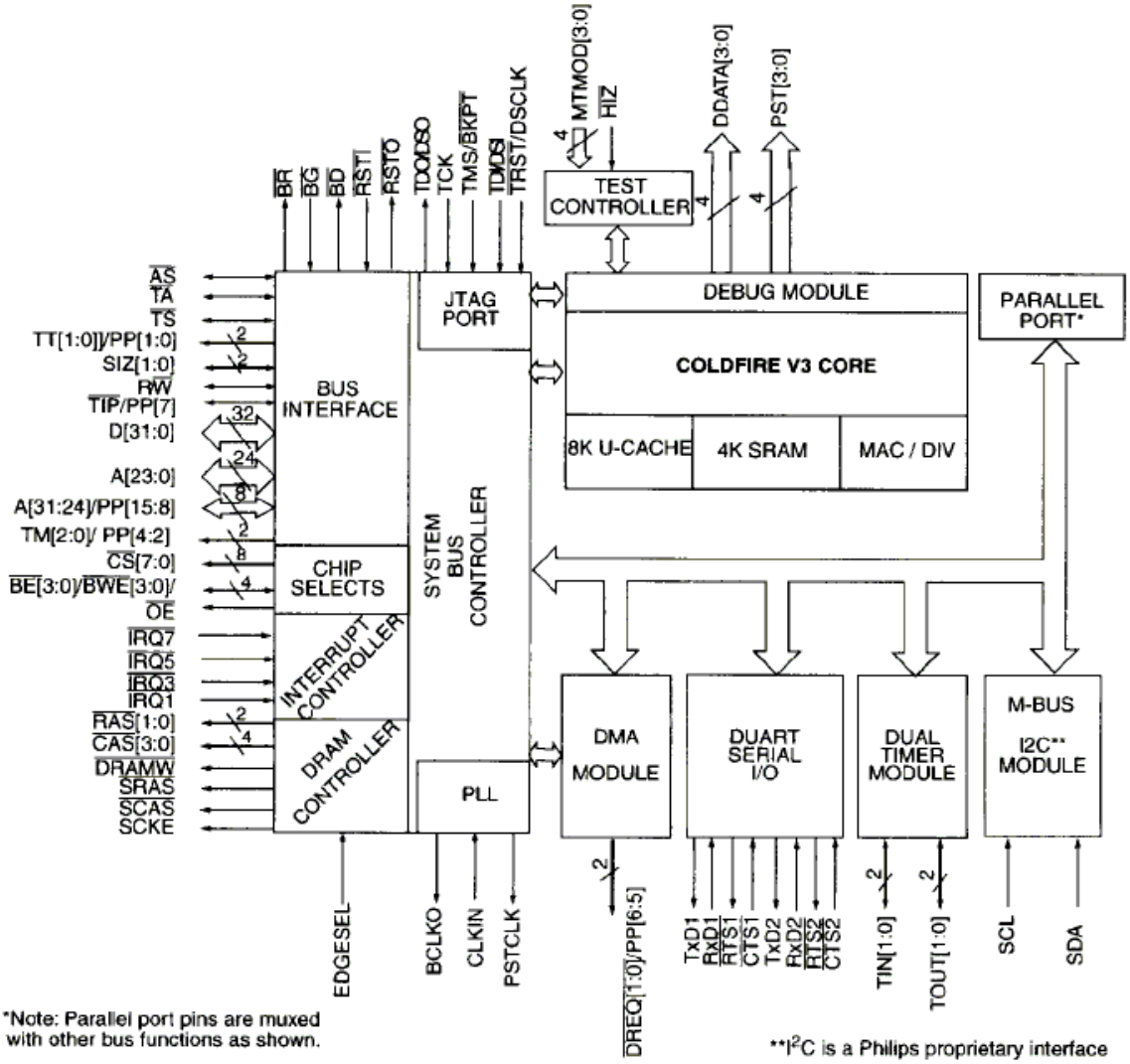

Figure 2. Cold Fire MCF5307 Block Diagram

Each of the functional blocks was considered in the design of the test software. An outline of each block and how that block is addressed in the test software is show in Table 1.

Each of the tests will be performed to identify any problems if at all possible. Therefore, the tests are carried out in a specific order to insure that errors that may occur will be pinpointed to the block in error. Complete details of each test is contained in Appendix A.

There are several blocks of the CPU that are instrumental in simply running code. These blocks are extremely difficult to test specifically but the fact that the code runs at all is evidence that these blocks are working.

The following blocks are considered operational if the test code runs.

- Processor core (ALU, pipeline)
- Address and data registers
- PLL circuit
- DRAM controller
- UART (serial output)
- Background Debug Module

These parts will not require testing but there proper operation will be verified in the execution of the test code.

Table 1

The functional blocks of the MCF5307 and the test software used to test each block

Block label	Test software
ColdFire V3 Core	Exercised in execution of test software. ALU and registers tested in a specific test. Floating-point operations also tested in software although this functionality is emulated in software.
8K U-Cache	Tested in specialized cache test. Exercises all cache modes (write-though, copy-back, write-buffer, etc.) and all read / write and hit / miss cases.
4K Internal SRAM	Tested in specialized SRAM test. Not used to store code or data during the duration of the test.
MAC / DIV	Exercised in ALU and floating-point test.
Debug Module	Background debug used to download and run code.
Test Controller	Not tested. Test controller not needed in space
JTAG Port	Not tested. JTAG port will not needed in space
PLL	Exercised in clock generation and accessing external memory.
Bus Interface	Exercised in accessing code and data from memory.
Chip Selects	Exercised while accessing Flash, DRAM and External SRAM.
Interrupt Controller	Exercised in software watchdog and DMA test.
DRAM Controller	Exercised while running code and storing data.
DMA Module	Tested in specialized DMA test.
DUART Serial I / O	Exercised in serial communications while testing.
Dual Timer Module	Tested in specialized timer test.
I2C Module	Not tested. It is would be difficult to implement a test procedure for this module given the required test setup.
Parallel Port	Not tested. It is would be difficult so implement a test procedure for this module given the required test setup.
Software Watchdog	Tested in safeguarding the system.

V. Test Methods

The normal process flow for testing the processor was to first perform a cold boot to ensure that the processor is in a known state. Then the test software is executed and allowed to perform a number of loops through all of the tests to check for proper operation of the processor. If this was successful, the beam would then be turned on and the processor monitor would be watched for events. If an event occurred that tripped the watchdog timer or completely froze the system a Single Event Functional Interrupt (SEFI)), the beam was stopped and the event recorded. Until that time, any error events that occurred were recorded and the processor and proton exposure were allowed to continue. At the end of a run (which ended in a SEFI), the processor was warm-booted to try to recover without a power cycle. When that would not recover the processor, a power cycle and reboot were done and the system checked out for the next run.

For proton testing, this cycle was repeated until the one device tested failed, presumably from total dose. For heavy ion testing, angles of 30, 45 and 60 degrees were used with the four ion beams to give 13 LETs across the LET range from 2.8 to over 100 MeV-cm^2/mg. Since total dose is generally not an issue for heavy ion testing (and was not for this device), one device was used to map the upset and SEFI characteristics as a function of LET. All four devices were then used, to ensure sufficient statistics and understanding device-to-device variations, it investigate device latchup at high LETs.

VI. Results

Proton Results

Proton-induced Single Event Upsets (SEU) and Single Event Functional Interrupts (SEFI) were seen on this part. SEUs were observed through Address errors, Floating Point Units (FPU) errors, etc. Samples of the types of SEU events that were seen are detailed in a summary of all proton results given in Table II. However, accurate estimate of their cross sections was overshadowed by the SEFI event rate. Therefore, only SEFI data will be presented in this report.

There were two types of SEFI events. The first is the case where the software watchdog timer was able to detect a stoppage in the main program operation. The second was where the processor operation was not detected by the watchdog, but by the operator noticed the SEFI and after stopping the beam, the debug module was used to reset the processor. Each test run ended in either one of these two types of events, with the exception of one run that was stopped prematurely. There were no SEFI events observed that required a power cycle to recover normal operations.

Therefore, the total SEFI cross section will be defined as the sum of these two types of events and the watchdog SEFI cross section will just be those events that the watchdog timer caught the problem. There were 28 total SEFI events detected. On the 29th run, total dose failure was observed (processor current had increased from approximately 750 mA to 1200 mA). Of those 28, six events were caught by the watchdog timer and a warm reboot of the processor recovered normal operation. Of the remaining 22 events, thirteen events required that the operator use the debug module to reset the processor.

The average cross sections for these SEFI events are given in Table III.

Table II Chronology of proton test results		
Run	**Encountered Problem**	**Outcome**
1	CPU froze while filling memory	Debug module restored operation
2	CPU froze while filling memory	Debug module restored operation
3	CPU froze during Internal SRAM test	Recovered via Software Watchdog Timer (SWT) soft reset
4	Error accessing Internal SRAM	SWT terminated bus cycle and restored proper operation
5	CPU froze while filling memory	Recovered via SWT soft reset
6	CPU froze during Internal SRAM test	Debug module restored operation
7	CPU froze while filling memory	Debug module restored operation
8	Persisting errors in Internal SRAM test	Debug module restored proper operation
9	CPU froze during External SRAM test	Debug module restored operation
10	Failure in Interrupt Controller during DMA test	Debug module restored operation
11	CPU froze during External SRAM test	Debug module restored operation
12	CPU froze during Internal SRAM test	Debug module restored operation
13	Single event error during Internal SRAM test	SWT terminated bus cycle and restored proper operation
14	CPU froze during Internal SRAM test	Debug module restored operation
15	CPU froze, illegal instruction	Debug module restored operation
16	CPU froze	Debug module restored operation
17	Persisting errors in Internal SRAM test	Debug module restored proper operation
18	Persisting errors in Cache test	Debug module restored proper operation
19	CPU froze during DMA test	Debug module restored operation
20	CPU froze	Debug module restored operation
21	Persisting errors in UART module	Debug module restored operation
22	CPU froze	Debug module restored operation
23	CPU froze during Internal SRAM test followed by illegal instruction error	Debug module restored operation
24	Single event error during Internal SRAM test	SWT terminated bus cycle and restored proper operation
25	CPU froze while filling memory	Debug module restored operation
26	CPU froze, illegal instruction	Debug module restored operation
27	CPU froze while filling memory	Recovered via SWT soft reset
28	Single event error in DMA test	Single event
29	CPU froze	Failure

Table III SEFI Results Summary		
Total SEFI Average Cross Section (cm^2)	Watchdog SEFI Average Cross Section (cm^2)	Debugger Reset SEFI Average Cross Section (cm^2)
2.45×10^{-11}	5.25×10^{-12}	1.93×10^{-11}

As stated above, in all but the last run, the processor could recover normal operation. In this last run after the SEFI event, normal operation could not be restored after any

number of power cycles and/or "cooling off" periods. As there was only one part to be tested, the testing of the ColdFire processor ended at this point.

It is probable that the failure was simply due to total dose buildup, as evidenced by the large increase in supply current. There was no evidence, however, on any previous runs that the processor operation was degraded. Other state-of-the-art processors have demonstrated this same normal operation to complete failure from total dose. Therefore, if this is considered a total dose failure, then the single part failure level was approximately 62 krads(Si).

Heavy Ion Results

Similar to the proton results Heavy Ion-induced Single Event Upsets (SEU) and Single Event Functional Interrupts (SEFI) were seen. However, the SEFI event rate from ions was so dominant that any measurement of upsets was impossible. Therefore, only SEFI data will be presented in this report.

SEFI measurements were taken across seven effective LET points through a range of 2.8 to 28.9 MeV-cm^2/mg. The results of these measurements are given in Figure 3. The solid black circles are the raw data points and the blue triangles are the averaged values (with the error bars representing one sigma of the average). A best fit to the Weibull function is done using these averages, giving and LET threshold of approximately 2.5 and a saturation cross section of approximately 2.9 x 10^{-5} cm^2.

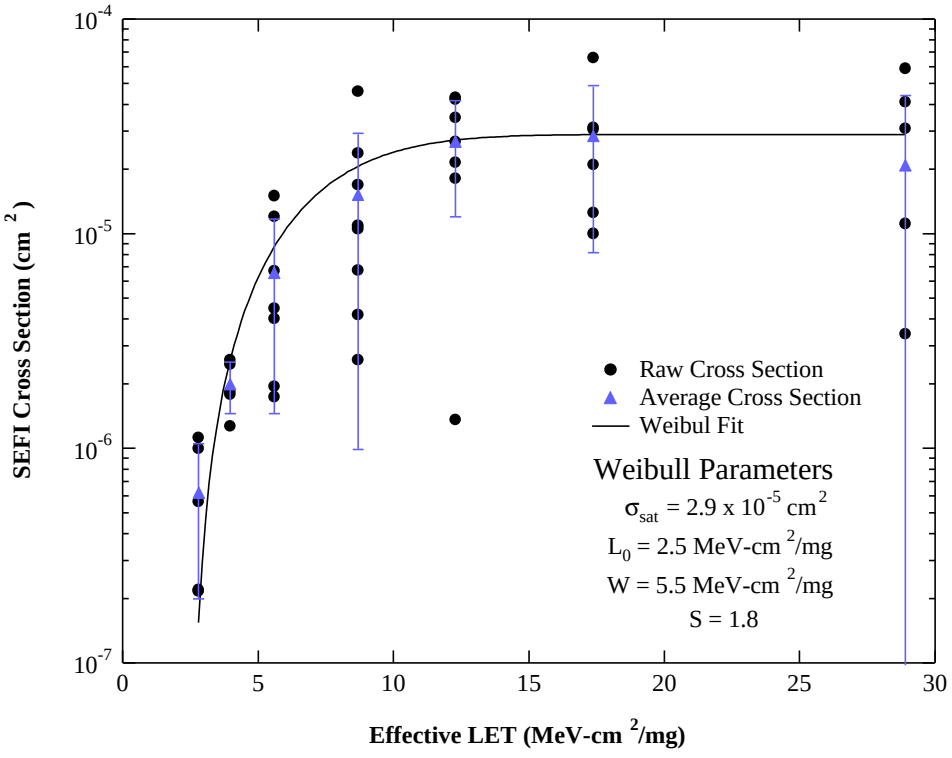

Figure 3. Heavy Ion-induced SEFI cross section as a function of Effective LET.

In addition to the SEFI testing, the ColdFire processor was evaluated for heavy ion-induced latchup. For this testing, there were 13 different effective LET values used ranging from 2.8 to 107 MeV-cm^2/mg. For each of these LET values a minimum fluence of 10^7 ions were incident on the four processors used in this testing. In all cases there was no indication of latchup in the processor. Therefore, the LET threshold for latchup for the ColdFire processor is greater than 107 MeV-cm^2/mg.

VII. Recommendations

In general, devices are categorized based on test data into one of the four following categories:

Category 1 – Recommended for usage in all NASA/GSFC spaceflight applications.
Category 2 – Recommended for usage in NASA/GSFC spaceflight applications, but may require mitigation techniques.
Category 3 – Recommended for usage in some NASA/GSFC spaceflight applications, but requires extensive mitigation techniques or hard failure recovery mode.
Category 4 – Not recommended for usage in any NASA/GSFC spaceflight applications.

Due to the high SEFI event rate, the Motorola ColdFire (MCF5307) Processors are Category 3 devices.

Appendix A

Register/ALU test

This test was designed to test both register loads/stores and basic ALU functions. Although the registers in the ColdFire will be tested inherently, specific testing eliminates any guesswork as to what is happening in the CPU.

The test will assign five variables to different registers and do several arithmetic and boolean operations on the values to verify correct operation of the ALU. It is not certain which registers are actually exercised, but based on the simple nature of this test program, generally, register r0-r4 are tested specifically.

The function that carries out the register test is given below.

```
int RegisterTest(void) {
      register int a,b,c,d,e,f;
      // Test immediate addressing
      a=0xAAAAAAAA;
      // Test immediate addition and register transfer
      b=a+0x11111111;
      // Test various boolean functions
      c=b<<1;
      d=~c;
      e=d&0x55555555;
      f=(c+e)|d;

      // Test for correctness
      if (a != 0xAAAAAAAA
            || b != 0xBBBBBBBB
            || c != 0x77777776
            || d != 0x88888889
            || e != 0x1
            || f != 0xFFFFFFFF
      ) {
            // Failed test
            printf("Register Test: Failed.\r\n");
            printf("Aborting further testing.\r\n");
            return 0;
      }
      printf("Register Test: Passed.\r\n");
      fflush(stdout);
      return 1;
}
```

Internal SRAM test

The ColdFire contains 8 kB of internal SRAM that is accessible in one clock cycle. This on-chip memory is easily tested by enabling it and writing know values to and from the SRAM at different locations.

The code used to test the internal SRAM is shown below.

```
int SRAMTest(void){
      unsigned long int temp1, temp2, temp3, temp4;
      unsigned long int currentAddress;

      /*    Internal SRAM test      */
      // Enable SRAM (set RAMBAR[V] = 1)
      cpu_iowr_32(RAMBAR, RAMBA | 0x1);

      // Begin test
      currentAddress = RAMBA;
      while (currentAddress < RAMBA+4096) {
            //    Write zeroes
            cpu_iowr_trap_32(currentAddress, 0x00000000);
            temp1 = cpu_iord_trap_32(currentAddress);
            //    Write ones
            cpu_iowr_trap_32(currentAddress, 0xFFFFFFFF);
            temp2 = cpu_iord_trap_32(currentAddress);
            //    Write Checkerboard
            cpu_iowr_trap_32(currentAddress, 0xAAAAAAAA);
            temp3 = cpu_iord_32(currentAddress);
            //    Invert Checkerboard
            cpu_iowr_trap_32(currentAddress, 0x55555555);
            temp4 = cpu_iord_trap_32(currentAddress);
            //    Verify correctness
            if (temp1 != 0x00000000 || temp2 != 0xFFFFFFFF ||
                  temp3 != 0xAAAAAAAA || temp4 != 0x55555555)
            {
                  printf("Memory Test: Failed.\r\n");
                  printf("Read/Write from Internal SRAM failed at
address %X.\r\n", currentAddress);
                  printf("Aborting further testing.\r\n");
                  return 0;
            }
            //    Test different portion of Internal SRAM
            currentAddress = currentAddress + 64;
      }
      printf("Memory Test: Passed.\r\n"); fflush(stdout);
      return 1;
}
```

Cache test

The MCF5307 feature a 4 kB unified cache. The cache is organized as 4-way set associative with a line size of 16 bytes. This means that each 32-bit address contains a 4-bit offset, 7-bit set field and a 21-bit tag field. The cache supports both write-through and copy-back modes as well as incorporating an optional 4 entry write buffer. All of these options were tested.

The test works as follows. The onboard external SRAM will be assigned to address 0x20000000 to separate data and code space. The external SRAM will be filled with psuedo-random values generated from the timer unit. The test program will then enable cache and verify that the correct values are properly read and written from the cachable memory space. The test will be carried out using each available mode.

Success of this test relies on enabling/disabling the cache and knowledge of the exact contents of the cache at any given time.

The code used for testing the code is show below.

```
int CacheTest(void) {
      unsigned char *mem;     // Memory space the size of cache
      unsigned long int i;

      //    Use ESRAMBA as cacheable space ensuring no code/data mixing
      mem = (unsigned char *)ESRAMBA;
      //    Set Timer to generate random values (reference to 256)
      cpu_iowr_16(TRR, 0xFFFF);
            //    Enable Timer
      cpu_iowr_16(TMR, 0x00F3);

            // Fill memory with alternating pattern of bits
      printf("Filling memory\n"); fflush(stdout);
      for (i = 0; i < ESRAMSIZE; i+=4) {
            // Load with pseudo-random values
            fillLongWord(((unsigned long int)mem)+i,
                  cpu_iord_8(TCN),
                  cpu_iord_8(TCN)+1,
                  cpu_iord_8(TCN)+2,
                  cpu_iord_8(TCN)+3);
            if (i%(ESRAMSIZE/64) == 0) {
                  printf("Filling memory %X/%X\n",i,ESRAMSIZE);
                  fflush(stdout);
            }
      }
      printf("                          \n");

      //    Run cache tests in standard mode
      cpu_iowr_32(ACR0, ESRAMBA | MCF5307_ACR_E |
                  MCF5307_ACR_S_USER | MCF5307_ACR_CM_00);
      if (CacheTestRun(mem)) {
            printf("Cache test passed in standard mode.\r\n");
            //    Enable Copy-back mode
            cpu_iowr_32(ACR0, cpu_iord_32(ACR0) | MCF5307_ACR_CM_01);
```

```
            if(CacheTestRun(mem)) {
                  printf("Cache test passed in copy-back mode.\r\n");
                  //    Enable store buffer
                  cpu_iowr_32(CACR, cpu_iord_32(CACR) | MCF5307_CACR_DNFB);

                  if(CacheTestRun(mem)) {
                        printf("Cache test passed store buffer.\r\n");
                  }
            }
      }
      //    Disable cache
      printf("Cache Test: Passed.           \r\n");  fflush(stdout);
      return 1;
}
int CacheTestRun(unsigned char *mem, unsigned long int cacr_val) {
      unsigned char temp[16];
      unsigned long int i;

      //    Disable to load reference array
      mcf5307_wr_cacr(0x00000000);

      /*    Load Reference Array    */
            // Set of 0
      temp[0]  = mem[0x00000];      // Tag 0
      temp[1]  = mem[0x00805];      // Tag 1
      temp[2]  = mem[0x01009];      // Tag 2
      temp[3]  = mem[0x0180F];      // Tag 3
            // Set of 16
      temp[4]  = mem[0x10101];      // Tag 50
      temp[5]  = mem[0x34103];      // Tag 100
      temp[6]  = mem[0x4B10A];      // Tag 150
      temp[7]  = mem[0x6410E];      // Tag 200
            // Set of 56
      temp[8]  = mem[0x08382];      // Tag 16
      temp[9]  = mem[0x10384];      // Tag 32
      temp[10] = mem[0x20387];      // Tag 64
      temp[11] = mem[0x4038C];      // Tag 128
            //    Set of 65
      temp[12] = mem[0x03C13];      // Tag 7
      temp[13] = mem[0x18C16];      // Tag 49
      temp[14] = mem[0x28C18];      // Tag 81
      temp[15] = mem[0x3CC1B];      // Tag 121

            //    Enable Cache
      mcf5307_wr_cacr(MCF5307_CACR_CINVA);
      mcf5307_wr_cacr(cacr_val);

      /*    Load 4 different cache lines all 4-ways         */
            // Load Set 0
      fillCacheLine(mem+0x0000);    //    Tag 0
      fillCacheLine(mem+0x0800);    //    Tag 1
      fillCacheLine(mem+0x1000);    //    Tag 2
      fillCacheLine(mem+0x1800);    //    Tag 3
            // Load Set 16
      fillCacheLine(mem+0x10100);   //    Tag 50
      fillCacheLine(mem+0x34100);   //    Tag 100
      fillCacheLine(mem+0x4B100);   //    Tag 150
      fillCacheLine(mem+0x64100);   //    Tag 200
            // Load Set 56
```

```
fillCacheLine(mem+0x08380);   //    Tag 16
fillCacheLine(mem+0x10380);   //    Tag 32
fillCacheLine(mem+0x20380);   //    Tag 64
fillCacheLine(mem+0x40380);   //    Tag 128
      // Load Set 65
fillCacheLine(mem+0x03C10);   //    Tag 7
fillCacheLine(mem+0x18C10);   //    Tag 49
fillCacheLine(mem+0x28C10);   //    Tag 81
fillCacheLine(mem+0x3CC10);   //  Tag 121

/*    Create read hits (by accessing bytes from the lines   */
      // Read from Set 0
if (mem[0x0000] != temp[0] || mem[0x0805] != temp[1] ||
      mem[0x1009] != temp[2] || mem[0x180F] != temp[3]) {
      cprintf("Cache read hit failed.\r\n");
      return 0;
}
      // Read from Set 16
if (mem[0x10101] != temp[4] || mem[0x34103] != temp[5] ||
      mem[0x4B10A] != temp[6] || mem[0x6410E] != temp[7]) {
      cprintf("Cache read hit failed.\r\n");
      return 0;
}
      // Read from Set 56
if (mem[0x08382] != temp[8] || mem[0x10384] != temp[9] ||
      mem[0x20387] != temp[10] || mem[0x4038C] != temp[11]) {
      cprintf("Cache read hit failed.\r\n");
      return 0;
}
      // Read from Set 65
if (mem[0x03C13] != temp[12] || mem[0x18C16] != temp[13] ||
      mem[0x28C18] != temp[14] || mem[0x3CC1B] != temp[15]) {
      cprintf("Cache read hit failed.\r\n");
      return 0;
}

/*    Create read misses      */
// Disable cache and read values
mcf5307_wr_cacr(0);

temp[0] = mem[0x2000];  //    Read from Set 0, Tag 4
temp[1] = mem[0x22900]; //    Read from Set 16, Tag 69
temp[2] = mem[0x4380];  //    Read from Set 56, Tag 8
temp[3] = mem[0x32410]; //    Read from Set 65, Tag 100

      //    Renable cache
mcf5307_wr_cacr(cacr_val);

      //    Create read misses
temp[0] = mem[0x2000];  //    Read from Set 0, Tag 4
temp[1] = mem[0x22900]; //    Read from Set 16, Tag 69
temp[2] = mem[0x4380];  //    Read from Set 56, Tag 8
temp[3] = mem[0x32410]; //    Read from Set 65, Tag 100

if (temp[0] != mem[0x2000] || temp[1] != mem[0x22900] ||
      temp[2] != mem[0x4380] || temp[3] != mem[0x32410]) {
      cprintf("Cache read miss failed.\r\n");
      return 0;
}
```

```
	/*    Create write hits */
	cpu_iowr_8(mem+0x2000,  0xAA);
	cpu_iowr_8(mem+0x22900, 0x55);
	cpu_iowr_8(mem+0x4380,  0xAB);
	cpu_iowr_8(mem+0x32410, 0x12);

	//    Test if written properly
	if (mem[0x2000] != 0xAA || mem[0x22900] != 0x55 ||
	      mem[0x4380] != 0xAB || mem[0x32410] != 0x12) {
	      cprintf("Cache write hit failed.\r\n");
	      return 0;
	}
	//    Flush values from cache
	      // Load Set 0
	fillCacheLine(mem+0x0000);    //    Tag 0
	fillCacheLine(mem+0x0800);    //    Tag 1
	fillCacheLine(mem+0x1000);    //    Tag 2
	fillCacheLine(mem+0x1800);    //    Tag 3
	      // Load Set 16
	fillCacheLine(mem+0x10100);   //    Tag 50
	fillCacheLine(mem+0x34100);   //    Tag 100
	fillCacheLine(mem+0x4B100);   //    Tag 150
	fillCacheLine(mem+0x64100);   //    Tag 200
	      // Load Set 56
	fillCacheLine(mem+0x08380);   //    Tag 16
	fillCacheLine(mem+0x10380);   //    Tag 32
	fillCacheLine(mem+0x20380);   //    Tag 64
	fillCacheLine(mem+0x40380);   //    Tag 128
	      // Load Set 65
	fillCacheLine(mem+0x03C10);   //    Tag 7
	fillCacheLine(mem+0x18C10);   //    Tag 49
	fillCacheLine(mem+0x28C10);   //    Tag 81
	fillCacheLine(mem+0x3CC10);   //  Tag 121

	      //    Disable cache and read values
	mcf5307_wr_cacr(0);

	if (mem[0x2000] != 0xAA || mem[0x22900] != 0x55 ||
	      mem[0x4380] != 0xAB || mem[0x32410] != 0x12) {
	      cprintf("Cache write hit failed w/ cache disabled.\r\n");
	      return 0;
	}
	      //    Renable cache
	mcf5307_wr_cacr(cacr_val);

	/*    Create write misses     */
	cpu_iowr_8(mem+0x2800, 0xA5); //    Set 0,      Tag 5
	cpu_iowr_8(mem+0x5000, 0x5A); //    Set 16, Tag 10
	cpu_iowr_8(mem+0x7800, 0xA5); //    Set 56, Tag 15
	cpu_iowr_8(mem+0xA000, 0x5A); //    Set 65, Tag 20

	if (mem[0x2800] != 0xA5 || mem[0x5000] != 0x5A ||
	      mem[0x7800] != 0xA5 || mem[0xA000] != 0x5A) {
	      cprintf("Cache write miss failed.\r\n");
	      return 0;
	}
	return 1;
}
```

Timer Unit Test

The TMU of the MCF5307 is tested using knowledge of the speed of the system clock and the speed of memory. The timer is set up to count from 0 to 255 continuously. The count is sampled and compared to the expected value plus and minus 2 counts.

The code for testing the Timer Unit is show below.

```
int TimerTest(void) {
      int i, j, e = 0;
      unsigned long int a, b, d;
            //    Set Timer reference to 256
      //cpu_iowr_16(TRR, 0xFFFF);
            //    Enable Timer
      cpu_iowr_16(TMR, 0x00F5);
      for(j = 0; j < 50; j++) {
            a = cpu_iord_16(TCN);
            for (i=0; i < 16; i++)
                  b = cpu_iord_16(TCN);
            if (b>a) d = b-a;
            else  d = (0xFFFF-a)+b;
            if (d < 51 || d > 53) {
                  e++;
            }
      }
      if (e > 2) {
            printf("Experienced %i errors in timer unit.\r\n", e);
            printf("Timer Test: Failed.\r\n");
            return 0;
      }
      printf("Timer Test: Passed.\r\n");  fflush(stdout);
      return 1;
}
```

DMA Test

The DMA unit of the MCF5307 contains four channels, only one of which is tested. The test code utilizes channel 0. Memory is filled with know values and then transferred to different location. The correct values are then verified.

The code for the DMA test is shown below.

```
int DMATest(void) {
      unsigned long int memSour[DMATransSize];
      unsigned long int memDest[DMATransSize];
      unsigned long int i;

      for (i = 0; i < DMATransSize; i++) {
            memSour[i] = i;
            memDest[i] = 0xDEADBEEF;
      }

      /*    Configure DMA     */
            //    Clear status register
      cpu_iowr_8(DSR0, 0x01);
            //    DMA0 Source Address Register to DRAM space
      cpu_iowr_32(SAR0, (unsigned long int)memSour);
            //    DMA0 Destination Address Register to internal SRAM
      cpu_iowr_32(DAR0, (unsigned long int)memDest);
            //    DMA0 BYTE count register to DMATransSize
      cpu_iowr_16(BCR0, 4*DMATransSize);
            //    Configure DMA interrupt
      cpu_iowr_8(ICR6, 0x1E);
                  //    Load DMAcomplete to location 0x100
      cpu_iowr_32(UDEV2, (unsigned long int)(DMAcomplete));
            //    DMA0 Interrupt Vector register to interrupt 65
      cpu_iowr_8(DIVR0, 65);
            //    Unmask the interrupt
      cpu_iowr_32(IMR, cpu_iord_32(IMR) & 0xFFFFBFFF);

      /*    Sets DMA Control Register to 0x8048 (BCR24BIT=0)      */
      cpu_iowr_16(DCR0, MCF5307_DMA_DCR_INT
            | MCF5307_DMA_DCR_SINC | MCF5307_DMA_DCR_SSIZE_LONG
            | MCF5307_DMA_DCR_DINC | MCF5307_DMA_DCR_DSIZE_LONG );

            //    Start DMA transfer
      cpu_iowr_16(DCR0, cpu_iord_16(DCR0) | MCF5307_DMA_DCR_START);

      while (DMADone == 0) {
            printf("Completed %i/%i of DMA transfer. Status: %X\n",
            DMATransSize-cpu_iord_16(BCR0),
            DMATransSize, cpu_iord_8(DSR0));
            fflush(stdout);
      }
            //    Disable DMA
      cpu_iowr_32(DCR0, 0x0);

      for (i = 0; i < DMATransSize; i++) {
            if (memSour[i] != memDest[i]) {
                  printf("DMA Test: Failed.\r\n");
```

```
                printf("Memory at offset %X did not transfer
                correctly.\r\n", i);
                printMemoryBlock((int*)(memSour+i-8), 16);
                printMemoryBlock((int*)(memDest+i-8), 16);
                return 0;
          }
      }

      printf("DMA Test: Passed.                              \r\n");
      fflush(stdout);
      return 1;
}
```

Floating-point test

The final test performed on the MCF5307 tests the floating-point functionality of the CPU and the multiply and accumulate feature. The code is straightforward and simply test the CPU results with know values.

The code for the floating-point test is shown below.

```
int FloatingPointTest(void) {
      double x, y, z;
      //    Test addition
      x = 3.14159;
      y = 2.718;
      z = x+y;
      if (z != 5.85959) {
            printf("Floating Point Test: Failed.\r\n");
            printf("Floaitng point addition failed.\r\n");
            printf("Aborting further testing.\r\n");
            return 0;
      }
      x = 3.5;
      y = 0.015625;
      z = x*y;
      if (z != 0.0546875) {
            printf("Floating Point Test: Failed.\r\n");
            printf("Floaitng point multiply failed.\r\n");
            printf("Aborting further testing.\r\n");
            return 0;
      }
      x = 4.9999;
      y = 5.0;
      if (x >= y) {
            printf("Floating Point Test: Failed.\r\n");
            printf("Floaitng point boolean failed.\r\n");
            printf("Aborting further testing.\r\n");
            return 0;
      }
      z = y/x;
      if (z != ((double)1.000020000400008000160003200064)) {
            printf("Floating Point Test: Failed.\r\n");
            printf("Floaitng point division failed.\r\n");
            printf("Aborting further testing.\r\n");
            return 0;
      }

      printf("Floating Point Test: Passed.\r\n");     fflush(stdout);
      return 1;
}
```